STOCKHOLM UNIVERSITY
Department of Statistics
Ellinor Fackle Fornius

Hand-in assignment 2

R Programming

This document contains R course material designed by Leif Johnsson, Måns Magnusson and Josef Wilzén

Contents

Instructions

The assignments have to be written in `R markdown` with the output in either `pdf` or `HTML` format. Both the `.Rmd` and `.pdf`/`.html`-files have to be handed in. Upload the documents to Athena no later than the deadline for this assignment (stated in the Course Description). Name your files with your group ID and assignment number. The report should include the code as well as the corresponding output. Comment your code when appropriate. For each task in the assignment you should reproduce the example code, when given. Otherwise, construct a relevant example of your own to illustrate the results.

1 `logical_equality()`

Create a function for "logical equal to" with the name `logical_equality()`. The function should have two arguments A and B,and return logical values in accordance with the following logical table:

A	B	logical_equality()
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE

```
logical_equality(A = TRUE, B = FALSE)
```

```
[1] FALSE
```

```
logical_equality(A = FALSE, B = FALSE)
```

```
[1] TRUE
```

2 `sheldon_game(player1, player2)`

In the series *The big bang theory* the character Sheldon comes up with a new version of the game Rock-paper-scissors, called Rock-paper-scissors-lizard-spock. The detailed rules of the game can be found **here**.

Create a function you call `sheldon_game()` that should take two arguments, `player1` and `player2`. Each player should choose one of the choices `rock`, `paper`, `scissors`, `lizard` or `spock`. If another choice is made, the function should be stopped (with the `stop()` or `stopifnot()` function).

The function should return either `"Player 1 wins!"`, `"Player 2 wins!"` or `"Draw!"`.

Hint! This can be solved by using the cyclical structure of the game and the modulus operator.

```
sheldon_game("lizard", "spock")

[1] "Player 1 wins!"

sheldon_game("rock", "paper")

[1] "Player 2 wins!"
```

3 my_moving_median()

Create a function that will create a rolling median from a given numeric vector using a while loop. Call the function `my_moving_median()`. The functions should accept a given numeric vector `x`, and a numeric scalar `n`.

The function should check if the arguments is a numeric vector (`x`) and a numeric scalar (`n`) and if not, it should stop the function (with the `stop()` or `stopifnot()` function). Otherwise the function should return the following vector:

$$y_t = \text{median}(x_t, \cdots, x_{t+n})$$

where y_t is the t th element. It should also be possible to send argument to the `median()` function (as `na.rm`).

```
my_moving_median(x = 1:10, n = 2)

[1] 2 3 4 5 6 7 8 9

my_moving_median(x = 5:15, n = 4)

[1]  7  8  9 10 11 12 13

my_moving_median(x = c(5, 1, 2, NA, 2, 5, 6, 8, 9, 9), n = 2)

[1]  2 NA NA NA  5  6  8  9

my_moving_median(x = c(5, 1, 2, NA, 2, 5, 6, 8, 9, 9), n = 2, na.rm = TRUE)

[1] 2.0 1.5 2.0 3.5 5.0 6.0 8.0 9.0
```

4 for_mult_table()

Write a function called `for_mult_table()` with arguments `from` and `to` using a for loop. Both arguments can be assumed to be positive integers. The function should return a multiplication table as a matrix from the argument `from` to the argument `to`. The column names and row names should contain the factors specified by the arguments. The function should stop (using `stop()`) if the arguments are not numeric scalars. See an example below.

```
for_mult_table(from = 1, to = 5)
```

```
  1  2  3  4  5
1 1  2  3  4  5
2 2  4  6  8 10
3 3  6  9 12 15
4 4  8 12 16 20
5 5 10 15 20 25
```

```
for_mult_table(from = 10, to = 12)
```

```
    10  11  12
10 100 110 120
11 110 121 132
12 120 132 144
```

5 cor_matrix()

Create a function called `cor_matrix()`. It should take a numeric `data.frame` with numerical variables of an arbitrary size as an argument `X` and calculate the correlation matrix for the given variables. If another object than a `data.frame` with numerical variables of an arbitrary size as an argument `is` used as argument, the function should return an error. See an example below.

In this exercise you are not allowed to use the functions `var()`, `sd()` or `cor()` in your function.

```
data(iris)
cor_matrix(iris[, 1:4])
```

```
         [,1]     [,2]     [,3]     [,4]
[1,]  1.00000 -0.11757  0.87175  0.81794
[2,] -0.11757  1.00000 -0.42844 -0.36613
[3,]  0.87175 -0.42844  1.00000  0.96287
[4,]  0.81794 -0.36613  0.96287  1.00000
```

```
data(faithful)
cor_matrix(faithful)
```

```
        [,1]    [,2]
[1,] 1.00000 0.90081
[2,] 0.90081 1.00000
```

6 find_cumsum()

Create a function that takes two arguments `x` and `find_sum`. The function should calculate the cumulative sum of the numeric vector `x` and stop when the cumulative sum is greater than `find_sum` or if the vector `x` is traversed and return this value. The function should assert that `x` is a numeric vector and `find_sum` is a numeric scalar (with stop() or stopifnot()).

It is not allowed to use a `for` loop in this assignment and it is not allowed to use the `cumsum()` function.

```
find_cumsum(x = 1:100, find_sum = 500)
```

```
[1] 528
```

```
find_cumsum(x = 1:10, find_sum = 1000)
```

```
[1] 55
```

7 while_mult_table()

Create a function that calulates a given mutiplication table similar to the exercise 4 but now you should use a (nested) while loop instead of a for loop. The result should be identical in all other respects.

```
while_mult_table(from = 3, to = 5)
```

```
   3  4  5
3  9 12 15
4 12 16 20
5 15 20 25
```

```
while_mult_table(from = 7, to = 12)
```

```
    7  8   9  10  11  12
```

```
7  49 56  63  70  77  84
8  56 64  72  80  88  96
9  63 72  81  90  99 108
10 70 80  90 100 110 120
11 77 88  99 110 121 132
12 84 96 108 120 132 144
```

8 repeat_find_cumsum()

Implement a function similar to `find_cumsum()` in 6 but using `repeat{}` instead of a `while` loop.

```
repeat_find_cumsum(x = 1:100, find_sum = 500)

[1] 528

repeat_find_cumsum(x = 1:10, find_sum = 1000)

[1] 55
```

9 repeat_my_moving_median()

Implement a function similar to `my_moving_median()` in 3 but using `repeat{}` instead of a `while` loop.

```
repeat_my_moving_median(x = 1:10, n = 2)

[1] 2 3 4 5 6 7 8 9

repeat_my_moving_median(x = 5:15, n = 4)

[1]  7  8  9 10 11 12 13

repeat_my_moving_median(x = c (5, 1, 2, NA, 2, 5, 6, 8, 9, 9), n = 2)

[1]  2 NA NA NA  5  6  8  9
```

10 coefvar()

Use `lapply()` and an anonymous function to find the coefficient of variation (the standard deviation divided by the mean) for all columns in a given `data.frame`. The function should

return a vector of the coefficients of variation. Assert that `X` is a `data.frame`.

```
data(iris)
coefvar(X = iris[1:4])

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
     0.14171      0.14256      0.46974      0.63555

coefvar(X = iris[3:4])

Petal.Length  Petal.Width
     0.46974      0.63555
```

11 bmi()

Write a function that you call `bmi()` with the arguments `body_weight` and `body_height`. The function must calculate BMI in the following way:

$$\texttt{bmi(body\_weight,body\_height)} = \frac{\texttt{body\_weight}}{\texttt{body\_height}^2}$$

and return the value. If `body_height` and/or `body_weight` is less than or equal to 0, the function has to **warn** that the current variable is less than or equal to 0 and the result is not meaningful:

`body_weight is not positive, calculation is not meaningful`

or

`body_height is not positive, calculation is not meaningful`

Test with different values for `body_length` and `body_weight`.

Here is an example of how the function should work:

```
bmi(body_weight = 63, body_height = 1.68)

[1] 22.321

bmi(body_weight = 95, body_height = 1.98)

[1] 24.232

myBMI <- bmi (body_weight = 95, body_height = 1.98)
myBMI

[1] 24.232

bmi(body_weight = 74, body_height = -1.83)
```

```
Warning in bmi(body_weight = 74, body_height = -1.83): body_height is not positive,
calculation is not meaningful

[1] 22.097

bmi(body_weight = 0, body_height = 1.63)

Warning in bmi(body_weight = 0, body_height = 1.63): body_weight is not positive,
calculation is not meaningful

[1] 0

bmi(body_weight = -73, body_height = 0)

Warning in bmi(body_weight = -73, body_height = 0): body_weight is not positive,
calculation is not meaningful
Warning in bmi(body_weight = -73, body_height = 0): body_height is not positive,
calculation is not meaningful

[1] -Inf

suppressWarnings(bmi(body_weight = -75, body_height = -1.64))

[1] -27.885
```

12 babylon()

An algorithm for approximating the square root of a number is the so called "Babylonian method". It's a way to calculate the square root of an arbitrary number.

The method, which is a special case of Newton-Raphson's method, can be described in the following way:

1. Start with an arbitrary proposal for the square root, called r_0. We need to start our algorithm at some point. The closer to the true the square root we start the fewer iterations will be needed.

2. Calculate a new proposal for the root as follows:

$$r_{n+1} = \frac{r_n + \frac{x}{r_n}}{2}$$

3. If $|r_{n+1} - r_n|$ has not reached arbitrary accuracy: go to step 2 again. **Note!** | | indicates the absolute amount of the difference between the iterations.

Implement this algorithm as a function in R. The function should be called `babylon()` and the arguments `x`, `init` and `tol`. `x` is the number for which the square root is to be approximated, `init` is the first proposal for the square root and `tol` is the accuracy required to terminate the algorithm. `tol=0.01` means that the algorithm should stop when $|r_{n+1} - r_n| \leq 0.01$.

The function can be implemented either as a `for` loop with `break` or a `while` loop. The function must return a list of two elements, `rot` and `iter` (both numeric values). In the element `rot`, the approximation of the square root returned and in the element `iter` the number of iterations must be returned.

Here is an example of how the algorithm should work: We want to calculate $\sqrt{10}$, our first guess is 3, so we put $r_0 = 3$, we select the tolerance level to 0.1.

1. We start by calculating $r_1 = \frac{r_0 + \frac{x}{r_0}}{2} = \frac{r_0 + 10/r_0}{2} = \frac{3+10/3}{2} = 3.166667$. Then we calculate the absolute difference between r_0 and r_1: $|r_1 - r_0| = |3.166667 - 3| = 0.166667$ since $0.166667 > 0.1$ we continue. This is the first iteration.

2. We calculate $r_2 = \frac{r_1 + \frac{x}{r_1}}{2} = \frac{3.166667 + 10/3.166667}{2} = 3.162281$. Then we calculate the absolute difference between r_1 and r_2: $|r_1 - r_2| = |3.166667 - 3.162281| = 0.004386$ since $0.004386 \leq 0.1$ then we interrupt the algorithm. This is the second interaction.

In this case, we got the final result $\sqrt{10} \approx 3.162281$ after two iterations.

Note! It is not allowed to use the function `sqrt()` in this task.

Here are examples of how the function should work:

```
test_list <-babylon(x = 40, init = 20, tol = 0.1)
test_list

$rot
[1] 6.3249

$iter
[1] 4

babylon(x = 2, init = 1.5, tol = 0.01)

$rot
[1] 1.4142

$iter
[1] 2

sqrt(2)

[1] 1.4142
```

```
babylon(x = 3, init = 2, tol = 0.000001)

$rot
[1] 1.7321

$iter
[1] 4

sqrt(3)

[1] 1.7321

babylon(x = 15, init = 1.5, tol = 0.01)

$rot
[1] 3.873

$iter
[1] 5

sqrt(15)

[1] 3.873
```

13 `hilbert_matrix()`

We will now create a function to produce arbitrarily large Hilbert matrices. Each element in a Hilbert matrix is defined as

$$H_{ij} = \frac{1}{i + j - 1} \tag{1}$$

Create a function `hilbert_matrix()` with the arguments `nrrow` and `nrcol`. The function should return a Hilbert matrix according to above. In this task it is practical to use a nested for loop. An example of the implementation follows:

1. Create a matrix H of the correct size
2. Create a nested for loop, where one loop goes over rows and one over columns in H.
3. Calculate 1 for each element in H using the loop index for the two loops.
4. Return H

In this task, it is not allowed to use built-in functions or R-packages that directly calculate Hilbert matrices, you have to create the function yourself according to the instructions above. Examples

of illicit functions are: `Hilbert()` in `Matrix`, `pbdDMAT`, `hilbert.matrix()` in `matrixcalc` and `hilb()` in `pracma`.

Below are examples of how the function should work.

```
hilbert_matrix(5, 2)

        [,1]    [,2]
[1,] 1.00000 0.50000
[2,] 0.50000 0.33333
[3,] 0.33333 0.25000
[4,] 0.25000 0.20000
[5,] 0.20000 0.16667

A <-hilbert_matrix (nrrow = 4, nrcol = 4)
A

        [,1]    [,2]    [,3]    [,4]
[1,] 1.00000 0.50000 0.33333 0.25000
[2,] 0.50000 0.33333 0.25000 0.20000
[3,] 0.33333 0.25000 0.20000 0.16667
[4,] 0.25000 0.20000 0.16667 0.14286

hilbert_matrix(3, 3)

        [,1]    [,2]    [,3]
[1,] 1.00000 0.50000 0.33333
[2,] 0.50000 0.33333 0.25000
[3,] 0.33333 0.25000 0.20000

hilbert_matrix(2, 3)

     [,1]    [,2]    [,3]
[1,]  1.0 0.50000 0.33333
[2,]  0.5 0.33333 0.25000
```

14 `toeplitz_matrix()`

Another special matrix is the so-called Toepliz matrix. It is also sometimes called a diagonal constant matrix. All diagonals has the same value. Below are two examples of Toeplitz matrices.

$$\begin{pmatrix} a & b & c & d \\ e & a & b & c \\ f & e & a & b \\ g & f & e & a \end{pmatrix}$$

$$\begin{pmatrix} 3 & 2 & 0 & 0 & 1 & 2 & 9 \\ 7 & 3 & 2 & 0 & 0 & 1 & 2 \\ 5 & 7 & 3 & 2 & 0 & 0 & 1 \\ 0 & 5 & 7 & 3 & 2 & 0 & 0 \\ 9 & 0 & 5 & 7 & 3 & 2 & 0 \\ 8 & 9 & 0 & 5 & 7 & 3 & 2 \\ 9 & 8 & 9 & 0 & 5 & 7 & 3 \end{pmatrix}$$

We will now create a function that creates a Toeplitz matrix of arbitrary size. It is possible to describe a Toeplitz matrix with a vector, let us call it x. x must be odd in length and give rise to a square matrix of size $n \times n$, where n is given by $n = \frac{length(x)+1}{2}$. To fully specify our Toeplitz matrix, we need to specify which elements in x belong to which diagonal in our Toeplitz matrix. Let us denote all diagonals as follows:

$$\begin{pmatrix} D_0 & D_{+1} & D_{+2} & \cdots & & \cdots & D_{+(n-1)} \\ D_{-1} & D_0 & D_{+1} & D_{+2} & & & \vdots \\ D_{-2} & D_{-1} & D_0 & D_{+1} & D_{+2} & & \\ \vdots & D_{-2} & D_{-1} & D_0 & D_{-1} & \ddots & \vdots \\ & & D_{-2} & D_{-1} & \ddots & \ddots & D_{+2} \\ \vdots & & & \ddots & \ddots & D_0 & D_{+1} \\ D_{-(n-1)} & \cdots & & \cdots & D_{-2} & D_{-1} & D_0 \end{pmatrix}$$

Here D_0 denotes the main diagonal. The index $+1$ means that that diagonal is closest to the main diagonal, $+2$ means the diagonal which is two steps above the main diagonal etc. Finally we have $D_{+(n-1)}$ which is the diagonal lying $(n-1)$ steps above the main diagonal, where n is the size of the matrix. In the same way, the index -1 means the diagonal closest to the main diagonal etc. We assume that x comes in the form $x = \begin{pmatrix} D_0 & D_{+1} & D_{+2} & D_{+3} & \cdots & D_{+(n-1)} & D_{-1} & D_{-2} & D_{-3} & \cdots & D_{-} \end{pmatrix}$ For example, $x = \begin{pmatrix} D_0 & D_{+1} & D_{+2} & D_{-1} & D_{-2} \end{pmatrix}$ gives rise to the matrix:

$$\begin{pmatrix} D_0 & D_{+1} & D_{+2} \\ D_{-1} & D_0 & D_{+1} \\ D_{-2} & D_{-1} & D_0 \end{pmatrix}$$

Another example is $x = \begin{pmatrix} D_0 & D_{+1} & D_{+2} & D_{+3} & D_{-1} & D_{-2} & D_{-3} \end{pmatrix}$ which gives the matrix:

$$\begin{pmatrix} D_0 & D_{+1} & D_{+2} & D_{+3} \\ D_{-1} & D_0 & D_{+1} & D_{+2} \\ D_{-2} & D_{-1} & D_0 & D_{+1} \\ D_{-3} & D_{-2} & D_{-1} & D_0 \end{pmatrix}$$

Now you will create a function that will take an arbitrary vector `x` and return a Toeplitz matrix based on this vector. The function should first check that the vector is of an odd length and otherwise the function should be stopped and the stop text "`x not of odd length`" printed. This task can be solved in several different ways. A suggestion for implementation comes here: (but you are welcome do in other ways!)

1. First check if `x` is of odd length, otherwise stop the function.
2. Create an empty matrix of the correct size, let's call it `res_mat`.
3. Change the order of the elements in `x`, if $x = \begin{pmatrix} D_0 & D_{+1} & D_{+2} & D_{-1} & D_{-2} \end{pmatrix}$ change so that $x = \begin{pmatrix} D_{-2} & D_{-1} & D_0 & D_{+1} & D_{+2} \end{pmatrix}$. So now D_0 is in the middle of the vector and the diagonals are arranged so that they come in ascending order seen from the center. **Hint!**`rev()`
4. Create a nested for loop that loops over rows and columns in `res_mat`. We now know that what value a diagonal should have depends on how many steps it is from the main diagonal (up or down). Use loop index from the two loops to calculate how many steps the current element is from the main diagonal and then select the correct value from `x` and save to the correct position in `res_mat.`
5. Return `res_mat.`

In this task, it is not allowed to use built-in functions or R-packages that directly calculate Toepliz matrices, you have to create the function itself according to the instructions above. Examples of illicit functions are: `toeplitz()` in `stats`, `toeplitz.spam()` in `spam`, `toeplitz()` in `ts`, `Toeplitz()` and `pracma`.

Below are examples of how the function should work.

```
toeplitz_matrix(x = "One element")
```

```
     [,1]
[1,] "One element"
```

```
toeplitz_matrix(x = 1:5)
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    1    2
[3,]    5    4    1
```

```
toeplitz_matrix(x = c(1, 4, 5, 2, 3))
```

```
     [,1] [,2] [,3]
[1,]    1    4    5
[2,]    2    1    4
[3,]    3    2    1
```

```
toeplitz_matrix(c("a", "b", "c", "d", "e", "f", "g"))
```

```
     [,1] [,2] [,3] [,4]
[1,] "a"  "b"  "c"  "d"
[2,] "e"  "a"  "b"  "c"
[3,] "f"  "e"  "a"  "b"
[4,] "g"  "f"  "e"  "a"
```

```
toeplitz_matrix(1:4)
```

```
Error in toeplitz_matrix(1:4): x not of odd length.
```

```
a <-toeplitz_matrix(c(1, 0, 0, 0, 0))
a
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

```
toeplitz_matrix(c(1, 0, 2, 0, 3, 0, 4, 0, 5))
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    2    0    3
[2,]    0    1    0    2    0
[3,]    4    0    1    0    2
[4,]    0    4    0    1    0
[5,]    5    0    4    0    1
```

```
toeplitz_matrix(x = c("D0", "D+1", "D+2", "D+3", "D-1", "D-2", "D-3"))
```

```
     [,1]  [,2]  [,3]  [,4]
[1,] "D0"  "D+1" "D+2" "D+3"
[2,] "D-1" "D0"  "D+1" "D+2"
[3,] "D-2" "D-1" "D0"  "D+1"
[4,] "D-3" "D-2" "D-1" "D0"
```